Debugger for Java

António Menezes Leitão

March, 2015

1 Introduction

In spite of the advances in compilers and typechecking, there is a considerable number of bugs that cannot be detected at compile-time. As a result, a significant fraction of the time spent debugging an application happens at runtime, i.e., during the execution of the application. However, detection of bugs at runtime can be an easier or harder tasks depending on the programming language that was used to write the application.

For example, in languages such as C, a bug can cause a malfunction in the execution of a program that can go undetected for quite some time, usually, causing additional problems until the program finally crashes, at a time where it might be very difficult to identify the bug. On the contrary, in the case of the Java programming language, the language was designed with considerable protection mechanisms that, for a large number of bugs, causes the JVM to throw an exception as soon as possible. Unfortunately, when that happens, the JVM debugging mechanisms are not powerful enough do allow the programmer to inspect the state of the program and, thus, more easily understand the bug.

2 Goals

Implementation of a *debugger* for Java programs. The debugger should, at load time, instrument a Java program so that, when an exception is thrown, the program is stopped and the programmer is presented with a command-line interface providing several debugging mechanisms.

The debugger must support the following set of commands:

- `Abort`: Terminates the execution of the application.
- `Info`: Presents detailed information about the called object, its fields, and the call stack. The presented information follows the format described in the next section.
- `Throw`: Re-throws the exception, so that it may be handled by the next handler.
- `Return <value>`: Ignores the exception and continues the execution of the application assuming that the current method call returned `<value>`. For calls to methods returning `void` the `<value>` is ignored. Note that `<value>` should be of a primitive type.
- `Get <field name>`: Reads the field named `<field name>` of the called object.
- `Set <field name> <new value>`: Writes the field named `<field name>` of the called object, assigning it the `<new value>`.
- `Retry`: Repeats the method call that was interrupted.

3 Example

As an example, consider the following Java classes defined in file `Example.java`:

```
1  package test;
2
3  class A {
4      int a = 1;
5
6      public double foo(B b) {
7          System.out.println("Inside A.foo");
8          if (a == 1) {
```

```
9               return b.bar(0);
10          } else {
11              return b.baz(null);
12          }
13      }
14  }
15
16  class B {
17      double b = 3.14;
18
19      public double bar(int x) {
20          System.out.println("Inside B.bar");
21          return (1/x);
22      }
23
24      public double baz(Object x) {
25          System.out.println("Inside B.baz");
26          System.out.println(x.toString());
27          return b;
28      }
29  }
30
31  public class Example {
32
33      public static void main(String[] args) {
34          System.out.println(new A().foo(new B()));
35      }
36  }
```

In order to execute the previous program using the debugger, the program must be started using the following form:

```
java ist.meic.pa.DebuggerCLI test.Example
```

As a result of the previous command, the program bytecode is modified at load-time and the program starts. If, for some reason, an exception is thrown, the program is interrupted, and the user is presented with the following prompt:

```
DebuggerCLI:>
```

It is now possible to inspect or modify the state of the program using the commands described previously. The next example explains the use of the previous commands.

Your assignment is to implement the class `ist.meic.pa.DebuggerCLI` which reads commands from the standard input and writes to standard error.

In order to implement the required output format, you should consider the following interaction example that was taken from a debugging session of the file `Example.java` described previously using the arguments `1 2 3`:

```
1  Inside A.foo
2  Inside B.bar
3  java.lang.ArithmeticException: / by zero
4  DebuggerCLI:> Info
5  Called Object:test.B@6cf5c440
6         Fields:b
7  Call stack:
8  test.B.bar(0)
9  test.A.foo(test.B@6cf5c440)
10 test.Example.main(1,2,3)
11 DebuggerCLI:> Get b
12 3.14
13 DebuggerCLI:> Throw
14 java.lang.ArithmeticException: / by zero
```

```
DebuggerCLI:> Info
Called Object:test.A@22726ef8
       Fields:a
Call stack:
test.A.foo(test.B@6cf5c440)
test.Example.main(1,2,3)
DebuggerCLI:> Set a 0
DebuggerCLI:> Retry
Inside A.foo
Inside B.baz
java.lang.NullPointerException
DebuggerCLI:> Info
Called Object:test.B@6cf5c440
       Fields:b
Call stack:
test.B.baz(null)
test.A.foo(test.B@6cf5c440)
test.Example.main(1,2,3)
DebuggerCLI:> Return 2
2.0
```

Note that the output of the Info command has the following syntax:

```
Called Object: <called object or null if static>
       Fields: <field1> ... <fieldN>
Call stack:
<called class>.<called method>(<arg1>,...,<argN>)
<called class>.<called method>(<arg1>,...,<argN>)
...
```

3.1 Extensions

You can extend your project to further increase your grade above 20. Note that this increase will not exceed **two** points that will be added to the project grade for the implementation of what was required in the other sections of this specification.

Be careful when implementing extensions, so that extra functionality does not compromise the functionality asked in the previous sections. In order to ensure this behavior, you should implement all your extensions in a different class named `ist.meic.pa.ExtendedDebuggerCLI`.

Some of the potentially interesting extensions include:

- Allow the use of non-primitive values in the `Set` command.
- Allow the use of non-primitive values in the `Return` command.
- Allow modification of the method call parameters before using `Retry`.
- Replace the method to be called with a different one.
- Replace the method to be called with a new block of code. The new block of code can access methods and fields just as if it was defined in the current object's class.

4 Code

Your implementation must work in Java 1.7 and should use the *bytecode* manipulation tool Javassist, version `3.19`.

The written code should have the best possible style, should allow easy reading and should not require excessive comments. It is always preferable to have clearer code with few comments than obscure code with lots of comments.

The code should be modular, divided in functionalities with specific and reduced responsibilities. Each module should have a short comment describing its purpose.

You must implement a Java class named `ist.meic.pa.DebuggerCLI` containing a static method `main` that accepts, as arguments, the name of another Java program (i.e., a Java class that also contains a static method

`main`)) and the arguments that should be provided to that program. The class should (1) operate the necessary transformations to the loaded Java classes so that the program can be debugged, and (2) should transfer the control to the `main` method of the program.

5 Format

Each project must be submitted by electronic means using the Fénix Portal. Each group must submit a single compressed file in ZIP format, named as `debugger.zip`. Decompressing this ZIP file must generate a folder named `g##`, where `##` is the group's number, containing:

- the source code, within subdirectory `/src`
- the slides of the presentation, in a file named `p1.pdf`.
- an Ant file `build.xml` file that, by default, compiles the code and generates `debugger.jar` in the same location where the file `build.xml` is located.

Note that it should be enough to execute

```
$ ant
```

to generate (`debugger.jar`). In particular, note that the submitted project must be able to be compiled when unziped and, as such, this means that you must include the file javassist.jar in the ZIP file.

The only accepted format for the presentation slides is PDF. This file must be located at the root of the ZIP file and must have the name `p1.pdf`

6 Evaluation

The evaluation criteria include:

- The quality of the developed solutions.
- The clarity of the developed programs.
- The quality of the public presentation.

In case of doubt, the teacher might request explanations about the inner workings of the developed project, including demonstrations.

The public presentation of the project is a compulsory evaluation moment. Absent students during project presentation will be graded zero in the entire project.

7 Plagiarism

It is considered plagiarism the use of any fragments of programs that were not provided by the teachers. It is not considered plagiarism the use of ideas given by colleagues as long as the proper attribution is provided.

This course has very strict rules regarding what is plagiarism. Any two projects where plagiarism is detected will receive a grade of zero.

These rules should not prevent the normal exchange of ideas between colleagues.

8 Final Notes

Don't forget Murphy's Law.

9 Deadlines

The code and the slides must be submitted via Fénix, no later than 23:00 of **March**, **30**.

The presentations will be done during the classes after the deadline. Only one element of the group will present the work and the presentation must not exceed 15 minutes. The element will be chosen by the teacher just before the presentation. Note that the grade assigned to the presentation affects the entire group and not only the person that will be presenting. Note also that content is more important than form. Finally, note that the teacher may question any member of the group before, during, and after the presentation.